XRay in LLVM

LLVM Developers' Meeting 2017
San Jose, CA, USA -- October 18, 2017

Dean Berris (@deanberris)

Why XRay?

Latency debugging in production is hard.

- Existing solutions use sampling to approximate sources of latency.
- Manual tracing is costly and error-prone.
- Some require OS-level support and introduce latency themselves.

Debugging with XRay

Use compiler-inserted instrumentation in functions to mark interesting event points.

Enable tracing dynamically through a set of APIs that can be controlled by the application.

Gather traces and analyse data offline.

Some Constraints

User-controllable instrumentation through attributes and command line flags.

Must work with LLVM IR.

User-pluggable architecture, to support non-tracing use-cases.

What is XRay?

XRay is compiler-inserted **instrumentation**.

XRay is an instrumentation **framework**.

XRay is a tracing **runtime**.

XRay is a set of **tools** for analysing traces.

XRay **instrumentation**.

XRay **framework**.

XRay **runtime**.

XRay **tools**.

XRay Instrumentation

Sleds and Maps

```
[[clang::xray_always_instrument]] void foo() { printf("Hello, XRay!\n"); }
```

Explicitly annotated functions for "always" or "never" instrumented functions.

clang++ **-fxray-instrument** ...

```
  .globl  _Z3foov
  .p2align  4, 0x90
  .type _Z3foov,@function
_Z3foov:                                # @_Z3foov
.Lfunc_begin0:
  .file 22 "test.cc"
  .loc  22 6 0                          # test.cc:6:0
  .cfi_startproc
# BB#0:                                 # %entry
  .p2align  1, 0x90
.Lxray_sled_0:
  .ascii  "\353\t"                      Entry sled.
  nopw  512(%rax,%rax)
.Ltmp0:
  pushq %rbp
.Ltmp1:
  .cfi_def_cfa_offset 16
.Ltmp2:
  .cfi_offset %rbp, -16
  movq  %rsp, %rbp
.Ltmp3:
  .cfi_def_cfa_register %rbp
  subq  $16, %rsp
  movabsq $.L.str, %rdi
.Ltmp4:
  .loc  22 6 48 prologue_end    # test.cc:6:48
  movb  $0, %al
  callq printf
  .loc  22 6 74 is_stmt 0       # test.cc:6:74
  movl  %eax, -4(%rbp)          # 4-byte Spill
  addq  $16, %rsp
  popq  %rbp
  .p2align  1, 0x90
.Lxray_sled_1:
  retq                                  Exit sled.
  nopw  %cs:512(%rax,%rax)
.Ltmp5:
.Lfunc_end0:
  .size _Z3foov, .Lfunc_end0-_Z3foov
  .cfi_endproc
```

```
; Function Attrs: uwtable
define void @_Z3foov() #0 {
entry:
  %call = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str, i32 0, i32 0))
  ret void
}
```

```
attributes #0 = { uwtable "disable-tail-calls"="false" "function-instrument"="xray-always" "less-precise-fpmad"="false" "no-frame-pointer-elim"="
true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"
="8" "target-cpu"="x86-64" "target-features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }
```

"function-instrument" is the keyword attribute coming from source-level attributes in C/C++, may be either **"xray-always"** or **"xray-never"**.

automatic function size heuristic comes in with **"xray-instruction-threshold"**="NNN".

```
  .p2align  4, 0x90
  .quad .Lxray_synthetic_0
  .section  xray_instr_map,"a",@progbits
.Lxray_synthetic_0:
  .quad .Lxray_sled_0
  .quad _Z3foov
  .byte 0
  .byte 1
  .zero 14
  .quad .Lxray_sled_1
  .quad _Z3foov
  .byte 1
  .byte 1
  .zero 14
  .text
```

Our XRay instrumentation map section.

Our entry sled for function "void foo()", which should always be instrumented.

Our exit sled for function "void foo()", which should always be instrumented.

XRay Instrumentation Heuristics

Only instrument machine functions with >= 200 machine instructions. Controllable by flags.

Instrument machine functions with loops.

Functions can be always/never instrumented through special-case lists, provided to Clang.

XRay Framework

Handlers and Patching

XRay Framework Constraints

Patching/Unpatching must not stop the process nor threads.

Installing/Uninstalling handlers must be atomic.

Only patch/unpatch instrumentation points.

```
  .p2align  4, 0x90
  .quad .Lxray_synthetic_0
  .section  xray_instr_map,"a",@progbits
.Lxray_synthetic_0:
  .quad .Lxray_sled_0
  .quad _Z3foov
  .byte 0
  .byte 1
  .zero 14
  .quad .Lxray_sled_1
  .quad _Z3foov
  .byte 1
  .byte 1
  .zero 14
  .text
```

Our XRay instrumentation map section.

Our entry sled for function "void foo()", which should always be instrumented.

Our exit sled for function "void foo()", which should always be instrumented.

Remember this?

AKA the instrumentation map.

Patching

For each entry in the instrumentation map:

Compute a function id for each function, in appearance order.

Patch the entry sled to become:

Set scratch register to function id.

Call an entry trampoline.

Patch the exit sled to become:

Set scratch register to function id.

Jump to an exit trampoline.

```
(gdb) disassemble foo
Dump of assembler code for function foo():
   0x0000000000415320 <+0>:     jmp    0x41532b <foo()+11>
   0x0000000000415322 <+2>:     nopw   0x200(%rax,%rax,1)
   0x000000000041532b <+11>:    push   %rbp
   0x000000000041532c <+12>:    mov    %rsp,%rbp
   0x000000000041532f <+15>:    sub    $0x10,%rsp
   0x0000000000415333 <+19>:    movabs $0x41ab2c,%rdi
   0x000000000041533d <+29>:    mov    $0x0,%al
   0x000000000041533f <+31>:    callq  0x401a50 <printf@plt>
   0x0000000000415344 <+36>:    mov    %eax,-0x4(%rbp)
   0x0000000000415347 <+39>:    add    $0x10,%rsp
   0x000000000041534b <+43>:    pop    %rbp
   0x000000000041534c <+44>:    retq
   0x000000000041534d <+45>:    nopw   %cs:0x200(%rax,%rax,1)
End of assembler dump.
```

```
(gdb) disassemble foo
Dump of assembler code for function foo():
   0x0000000000415320 <+0>:     mov    $0x1,%r10d
   0x0000000000415326 <+6>:     callq  0x4145e0 <__xray_FunctionEntry>
   0x000000000041532b <+11>:    push   %rbp
   0x000000000041532c <+12>:    mov    %rsp,%rbp
   0x000000000041532f <+15>:    sub    $0x10,%rsp
   0x0000000000415333 <+19>:    movabs $0x41ab2c,%rdi
   0x000000000041533d <+29>:    mov    $0x0,%al
   0x000000000041533f <+31>:    callq  0x401a50 <printf@plt>
   0x0000000000415344 <+36>:    mov    %eax,-0x4(%rbp)
   0x0000000000415347 <+39>:    add    $0x10,%rsp
   0x000000000041534b <+43>:    pop    %rbp
   0x000000000041534c <+44>:    mov    $0x1,%r10d
   0x0000000000415352 <+50>:    jmpq   0x4146d0 <__xray_FunctionExit>
End of assembler dump.
```

Thread-Safe Patching

Complete sequence to write:
`mov $0x1, %r10d` (6b)
`callq __xray_FunctionEntry` (5b)

11 bytes worth to overwrite:
`jmp +9` (2b)
`nopw 0x200(%rax,%rax,1)` (9b)

Solution:

- jmp +9 must be 2-byte aligned.
- Write 5 bytes of `callq __xray_FunctionEntry` first to last 5 bytes of sled.
- Write last 4 bytes of `mov N, %r10d` next to bytes 3-6.
- Atomically write 2 bytes of `mov N, %10d` over `jmp +9`.

```
mov     $0x1,%r10d
callq   0x4145e0 <__xray_FunctionEntry>
push    %rbp
mov     %rsp,%rbp
sub     $0x10,%rsp
movabs  $0x41ab2c,%rdi
mov     $0x0,%al
callq   0x401a50 <printf@plt>
mov     %eax,-0x4(%rbp)
add     $0x10,%rsp
pop     %rbp
mov     $0x1,%r10d
jmpq    0x4146d0 <__xray_FunctionExit>
```

Two-byte alignment is important because:

- Cache line is evenly-sized.
- Write over 2 bytes must appear atomic, must never straddle a cache line.
- Writes to other cache lines must happen before atomic write over the 2-byte jump.

Un-patching

For each entry in the instrumentation map:

Patch the entry sled to become:

jump across 9 bytes.

Patch the exit sled to become:

ret before 10 bytes.

We cannot safely, atomically restore the noops once we've patched the sleds in the presence of multiple threads.

Installing Handlers

- We can install any handler with `__xray_set_handler(...)`.
  - The handler only needs to take two arguments: function id (int32_t) and entry type (an enum type, i.e. int).
  - The handler has to be thread-safe, re-entrant, and signal-aware.
    - The handler can pretty much do what it wants. :)
- Install other handlers with `__xray_set_handler_arg1(...)` for capturing first argument of functions attributed to capture the first argument (useful for "this" pointer)
- Support for custom event logging with `__xray_set_customevent_handler(...)` for capturing custom events provided by `__xray_customevent(...)` clang built-in.

XRay Runtime

Tracing Implementation

Tracing Implementations

- Basic (naïve) mode.
  - Writes fixed-sized (32b) raw records to disk, one record per event (entry, exit), keeping a buffer per-thread.
- Flight Data Recorder (FDR) mode.
  - Writes variable sized (8-16b) records to disk, filtered by duration and type, using a circular buffer of fixed-sized-buffers handed out and returned as needed per-thread.

Both implementations cover the same events.

Logs written by both implementations readable by XRay tools.

Basic mode is great for command line tools that have short (O(seconds)) lifetimes. Generates a ton of data.

FDR mode is great for long-running applications, and for capturing short durations (O(seconds)).

Basic Mode Demo

Contrived Example

```cpp
#include <iostream>

[[clang::xray_always_instrument]] void foo() {
  std::cout << "Hello, XRay!" << std::endl;
}

[[clang::xray_always_instrument, clang::xray_log_args(1)]]
void bar(int i) {
  std::cout << "Captured: " << i << std::endl;
}

[[clang::xray_always_instrument]] int main(int argc, char* argv[]) {
  foo();
  bar(argc);
}
```

```
clang++ -fxray-instrument -std=c++11 \
hello_xray.cpp -o hello_xray
```

```
$ ./hello_xray
Hello, XRay!
Captured: 1
```

```
$ XRAY_OPTIONS="patch_premain=true xray_naive_log=true" ./hello_xray
==NNNNN==XRay: Log file in 'xray-log.hello_xray.xxxx'
Hello, XRay!
Captured: 1
```

We can analyse 'xray-log.hello_xray.xxxx' later.

FDR Mode Demo

Another Contrived Example

```
 1 #include <atomic>
 2 #include <cassert>
 3 #include <iostream>
 4 #include <random>
 5 #include <string>
 6 #include <thread>
 7 #include "time.h"
 8 #include "xray/xray_log_interface.h"
 9
10 constexpr auto kBufferSize = 16 * 1024;
11 constexpr auto kBufferMax = 10;
12
13 [[clang::xray_always_instrument]] void __attribute__((noinline)) bar(int num) {
14   if (num % 2) {                 // odd number.
15     const timespec t{0, 10000};  // 10 microsecs
16     timespec rem;
17     clock_nanosleep(CLOCK_REALTIME, 0, &t, &rem);
18   } else {                        // even number.
19     const timespec t{0, 100000};  // 100 microsecs
20     timespec rem;
21     clock_nanosleep(CLOCK_REALTIME, 0, &t, &rem);
22   }
23 }
24
25 std::atomic<bool> stopping{false};
26
27 [[clang::xray_always_instrument]] void __attribute__((noinline)) foo() {
28   static bool unused = [] {
29     std::srand(std::time(0));
30     return false;
31   }();
32   (void)unused;
33   while (!stopping.load(std::memory_order_relaxed)) {
34     bar(std::rand());
35   }
36 }
```

```
37
38 int main(int argc, char* argv[]) {
39   std::thread t1(foo);
40   std::thread t2(foo);
41
42   // Set up FDR mode.
43   using namespace __xray;
44   FDRLoggingOptions Options;
45   __xray_patch();
46   auto status = __xray_log_init(kBufferSize, kBufferMax, &Options,
47                                 sizeof(FDRLoggingOptions));
48   assert(status == XRayLogInitStatus::XRAY_LOG_INITIALIZED);
49   std::string input;
50   std::cin >> input;  // wait for input, keep threads running.
51   __xray_log_finalize();
52
53   // Wait for a while, to let threads see the finalization.
54   const timespec t{0, 100000};
55   timespec rem;
56   clock_nanosleep(CLOCK_REALTIME, 0, &t, &rem);
57
58   // Now flush the log.
59   __xray_log_flushLog();
60   __xray_unpatch();
61
62   // Stop the threads.
63   stopping.store(true, std::memory_order_release);
64   t1.join();
65   t2.join();
66 }
```

```
$ XRAY_OPTIONS="xray_naive_log=false xray_fdr_log=true" ./hello_world_fdr
==11228==XRay FDR init successful.
stop
==11228==XRay: Log file in 'xray-log.hello_world_fdr.yyyyyy'
```

We can analyse 'xray-log.hello_world_fdr.yyyyyy' later.

XRay Tools

Extract, Convert, Graph, Account, and Stack.

llvm-xray

Code in llvm/tools/llvm-xray/*

Libraries in {include,lib}/XRay/*

Sub-commands implement key functionality.

Extensions implemented as self-registering sub-commands.

llvm-xray extract

```
$ llvm-xray extract ./hello_xray
---
- { id: 1, address: 0x000000000041E7B0, function: 0x000000000041E7B0, kind: function-enter, always-instrument: true,
function-name: '' }
- { id: 1, address: 0x000000000041E7F8, function: 0x000000000041E7B0, kind: function-exit, always-instrument: true,
function-name: '' }
- { id: 2, address: 0x000000000041E810, function: 0x000000000041E810, kind: log-args-enter, always-instrument: true,
function-name: '' }
- { id: 2, address: 0x000000000041E868, function: 0x000000000041E810, kind: function-exit, always-instrument: true,
function-name: '' }
- { id: 3, address: 0x000000000041E880, function: 0x000000000041E880, kind: function-enter, always-instrument: true,
function-name: '' }
- { id: 3, address: 0x000000000041E8AE, function: 0x000000000041E880, kind: function-exit, always-instrument: true,
function-name: '' }
...

$ llvm-xray extract ./hello_xray --symbolize
---
- { id: 1, address: 0x000000000041E7B0, function: 0x000000000041E7B0, kind: function-enter, always-instrument: true,
function-name: 'foo()' }
- { id: 1, address: 0x000000000041E7F8, function: 0x000000000041E7B0, kind: function-exit, always-instrument: true,
function-name: 'foo()' }
- { id: 2, address: 0x000000000041E810, function: 0x000000000041E810, kind: log-args-enter, always-instrument: true,
function-name: 'bar(int)' }
- { id: 2, address: 0x000000000041E868, function: 0x000000000041E810, kind: function-exit, always-instrument: true,
function-name: 'bar(int)' }
- { id: 3, address: 0x000000000041E880, function: 0x000000000041E880, kind: function-enter, always-instrument: true,
function-name: main }
- { id: 3, address: 0x000000000041E8AE, function: 0x000000000041E880, kind: function-exit, always-instrument: true,
function-name: main }
...
```

llvm-xray convert

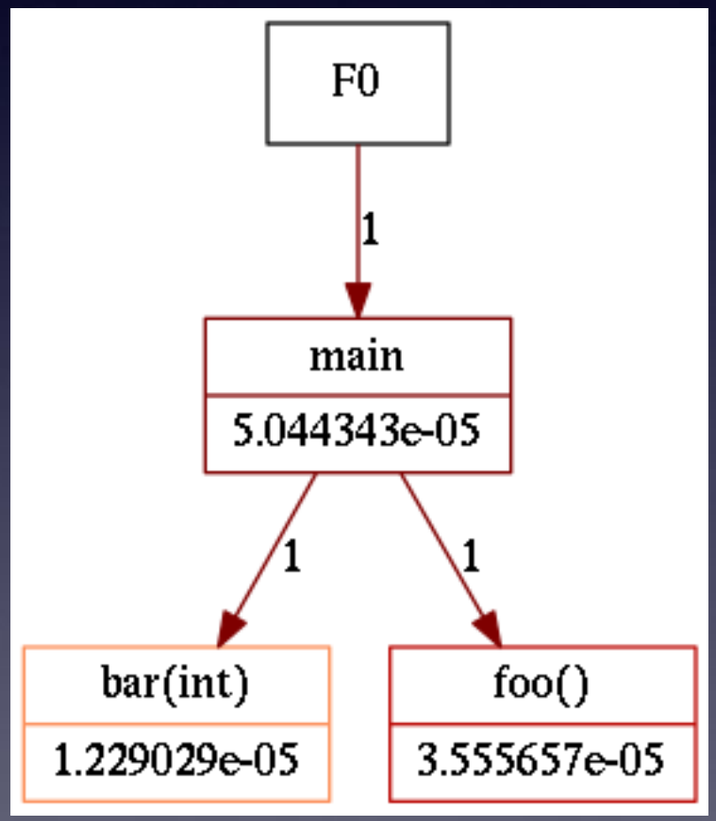
```
$ llvm-xray convert -instr_map=./hello_xray -output-format=yaml xray-log.hello_xray.xxxxxx
---
header:
  version:         2
  type:            0
  constant-tsc:    true
  nonstop-tsc:     true
  cycle-frequency: 3500000000
records:
  - { type: 0, func-id: 3, function: '3', cpu: 12, thread: 13520, kind: function-enter, tsc: 242080252932140
16 }
  - { type: 0, func-id: 1, function: '1', cpu: 12, thread: 13520, kind: function-enter, tsc: 242080252932220
88 }
  - { type: 0, func-id: 1, function: '1', cpu: 12, thread: 13520, kind: function-exit, tsc: 242080252933465
36 }
  - { type: 0, func-id: 2, function: '2', args: [ 1 ], cpu: 12, thread: 13520, kind: function-enter-arg, tsc:
 242080252933473
44 }
  - { type: 0, func-id: 2, function: '2', cpu: 12, thread: 13520, kind: function-exit, tsc: 242080252933903
60 }
  - { type: 0, func-id: 3, function: '3', cpu: 12, thread: 13520, kind: function-exit, tsc: 242080252933905
68 }
...
```

llvm-xray convert

```
$ llvm-xray convert -instr_map=./hello_world_fdr -output-format=yaml xray-log.hello_world_fdr.yyyyyy --symbolize | head
-20
---
header:
  version:         1
  type:            1
  constant-tsc:    true
  nonstop-tsc:     true
  cycle-frequency: 3500000000
records:
  - { type: 0, func-id: 1, function: 'bar(int)', cpu: 36, thread: 11229, kind: function-exit, tsc: 78474232425782 }
  - { type: 0, func-id: 1, function: 'bar(int)', cpu: 36, thread: 11229, kind: function-enter, tsc: 78474232435734 }
  - { type: 0, func-id: 1, function: 'bar(int)', cpu: 36, thread: 11229, kind: function-exit, tsc: 78474232838062 }
  - { type: 0, func-id: 1, function: 'bar(int)', cpu: 36, thread: 11229, kind: function-enter, tsc: 78474232840134 }
  - { type: 0, func-id: 1, function: 'bar(int)', cpu: 36, thread: 11229, kind: function-exit, tsc: 78474233009046 }
  - { type: 0, func-id: 1, function: 'bar(int)', cpu: 36, thread: 11229, kind: function-enter, tsc: 78474233011286 }
  - { type: 0, func-id: 1, function: 'bar(int)', cpu: 36, thread: 11229, kind: function-exit, tsc: 78474233180006 }
  - { type: 0, func-id: 1, function: 'bar(int)', cpu: 36, thread: 11229, kind: function-enter, tsc: 78474233181198 }
  - { type: 0, func-id: 1, function: 'bar(int)', cpu: 36, thread: 11229, kind: function-exit, tsc: 78474233583558 }
  - { type: 0, func-id: 1, function: 'bar(int)', cpu: 36, thread: 11229, kind: function-enter, tsc: 78474233585502 }
  - { type: 0, func-id: 1, function: 'bar(int)', cpu: 36, thread: 11229, kind: function-exit, tsc: 78474233987726 }
  - { type: 0, func-id: 1, function: 'bar(int)', cpu: 36, thread: 11229, kind: function-enter, tsc: 78474233989622 }
```

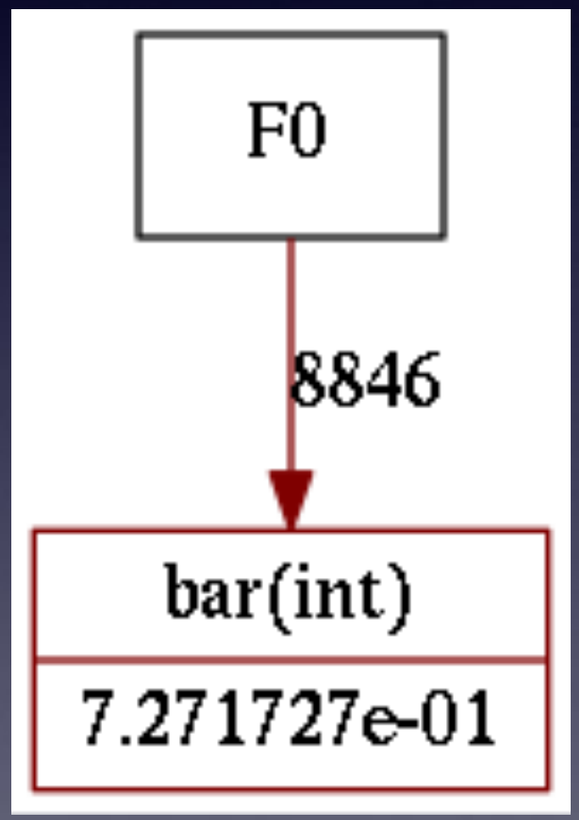
llvm-xray graph

```
$ llvm-xray graph -instr_map=./hello_xray xray-log.hello_xray.* -color-edges=count \
    -edge-label=count -color-vertices=sum -vertex-label=sum | dot -png -x > /tmp/hello_xray.png
```



llvm-xray graph

```
$ llvm-xray graph -instr_map=./hello_world_fdr xray-log.hello_world_fdr.* -color-edges=count \
    -edge-label=count -color-vertices=sum -vertex-label=sum --keep-going | \
    dot -png -x > /tmp/hello_world_fdr.png
```



llvm-xray account

```
$ llvm-xray account -instr_map=./hello_xray xray-log.hello_xray.xxxxxx
Functions with latencies: 3
   funcid      count [      min,      med,      90p,      99p,      max]      sum  function
   1           1 [ 0.000036,  0.000036,  0.000036,  0.000036,  0.000036]  0.000036  hello_xray.cpp:3:0: foo()
   2           1 [ 0.000012,  0.000012,  0.000012,  0.000012,  0.000012]  0.000012  hello_xray.cpp:7:0: bar(int)
   3           1 [ 0.000050,  0.000050,  0.000050,  0.000050,  0.000050]  0.000050  hello_xray.cpp:11:0: main
```

llvm-xray account

```
$ llvm-xray account --keep-going -instr_map=./hello_world_fdr xray-log.hello_world_fdr.yyyyyy 2>/dev/null
Functions with latencies: 1
  funcid      count [      min,       med,       90p,       99p,       max]       sum  function
       1       8846 [ 0.000016,  0.000114,  0.000115,  0.000121,  0.000192]  0.727258  hello_world_fdr.cpp:13:0: bar(int)
```

llvm-xray stack

```
./bin/llvm-xray stack -instr_map=./hello_world_fdr \
    xray-log.hello_world_fdr.s3c4ft --keep-going \
    --stack-format=flame -all-stacks 2>/dev/null | \
    ~/Source/FlameGraph/flamegraph.pl \
    > /tmp/hello_xray_flame.svg
```

FlameGraph tool from https://github.com/brendangregg/FlameGraph

Demo Flame Graphs

Recap

XRay is compiler-inserted **instrumentation**.

XRay is an instrumentation **framework**.

XRay is a tracing **runtime**.

XRay is a set of **tools** for analysing traces.

XRay **instrumentation**.

XRay **framework**.

XRay **runtime**.

XRay **tools**.

XRay availability (Linux)

feature	x86_64	ppc64le	arm	aarch64	mips
no-arg logging	✔	✔	✔	✔	✔
1-arg logging	✔	✔	✔	✔	✔
basic/naive mode	✔	✔	✔	✔	✔
custom events	✔				
flight data recorder mode	✔				

Future Work

Support more OSes.

Support more modes.

Add more tools for visualisation and analysis.

Thank you!

Contacts:
Dean Berris (@deanberris)
XRay Team (google-xray@googlegroups.com)